Stacks

CSE 2011

Fall 2009

Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations
- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

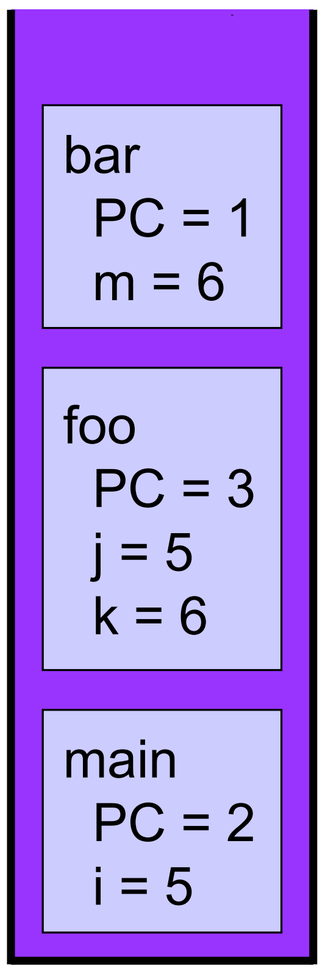
Stacks: LIFO

- Insertions and deletions follow the Last-In First-Out rule
- Applications, examples:
  - Undo operation in a text editor
  - History of visited web pages
  - Sequence of method calls in Java

Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack
- Allows for **recursion**

```
main() {
  int i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }

bar(int m) {
  ...
  }
```



Stack ADT

- Data stored: arbitrary objects
- Operations:
  - ***push***(object): inserts an element
  - object ***pop***(): removes and returns the last inserted element
- Other useful operations:
  - object ***top***(): returns the last inserted element without removing it

Error Conditions

- ***push***(object)
- object ***pop***()
- object ***top***()
- Exceptions are thrown when an operation cannot be executed.
- Execution of ***pop***() or ***top***() on an empty stack
  → throws *EmptyStackException.*
- Another useful operation:
  - **boolean *isEmpty***(): returns true if the stack is empty; false otherwise.

Stack Operations

- ***push***(object)
- object ***pop***()
- object ***top***()
- **boolean *isEmpty***()

- Still another useful operation:
  int *size*(): returns the number of elements in the stack

- Any others?
  Depending on implementation

Stack Interface in Java

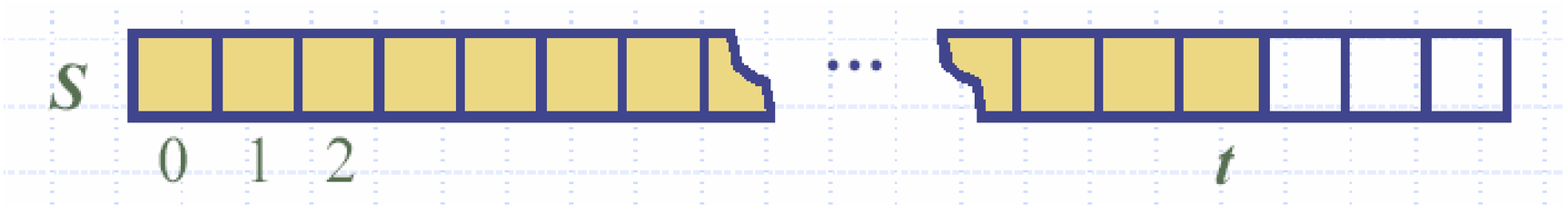
- Java interface corresponding to our Stack ADT
- Requires the definition of class EmptyStackException
- Different from the built-in Java class java.util.Stack

```
public interface Stack {

   public int size();

   public boolean isEmpty();

   public Object top()
         throws EmptyStackException;

   public void push(Object o);

   public Object pop()
         throws EmptyStackException;
}
```

Array-based Implementation

- An array *S* of maximum size *N*
- A variable *t* that keeps track of the top element in array *S*

- Top element: *S*[*t*]
- Stack is empty: ?
- Number of elements in the stack: ?



Pseudo-code

```
Algorithm size():
  return (t + 1);

Algorithm isEmpty():
  return (t < 0);

Algorithm top():
  if (isEmpty())
   throw StackEmptyException;
  return S[t];
```

```
Algorithm pop():
  if (isEmpty())
   throw StackEmptyException;
  temp = S[t];
  t = t – 1;
  return temp;
```

Optimization: set *S*[*t*] to *null* before decrementing *t*

Homework: implement *pop*() without any temp variable

Method ***push***()

Algorithm ***push***(object):
  $t = t + 1$;
 $S[t]$ = object;

- The array may become full
- ***push***() method will then throw a *FullStackException*
- Limitation of array-based implementation

Algorithm ***push***(object):
  if (*size*() == *N*)
    throw *FullStackException*;
  $t = t + 1$;
 $S[t]$ = object;

Array-based Stack in Java

```
public class ArrayStack
      implements Stack {

   // holds the stack elements
   private Object S[ ];

   // index to top element
   private int top = -1;

   // constructor
   public ArrayStack(int capacity) {
       S = new Object[capacity]);
    }
```

```
   public Object pop()
         throws EmptyStackException {
      if isEmpty()
         throw new EmptyStackException
            (“Empty stack: cannot pop”);
      Object temp = S[top];
      // facilitates garbage collection
      S[top] = null;
      top = top – 1;
      return temp;
   }
```

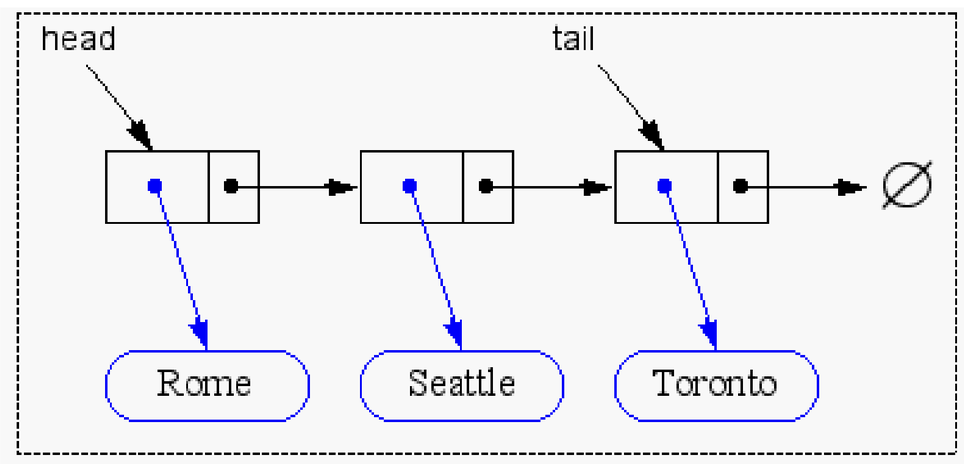
Performance of Array Implementation

- Each operation runs in $O(1)$ time
  (no loops, no recursion)
- Array-based implementation is simple, efficient, but …
- The maximum size N of the stack is fixed
- How to determine N? Not easy!

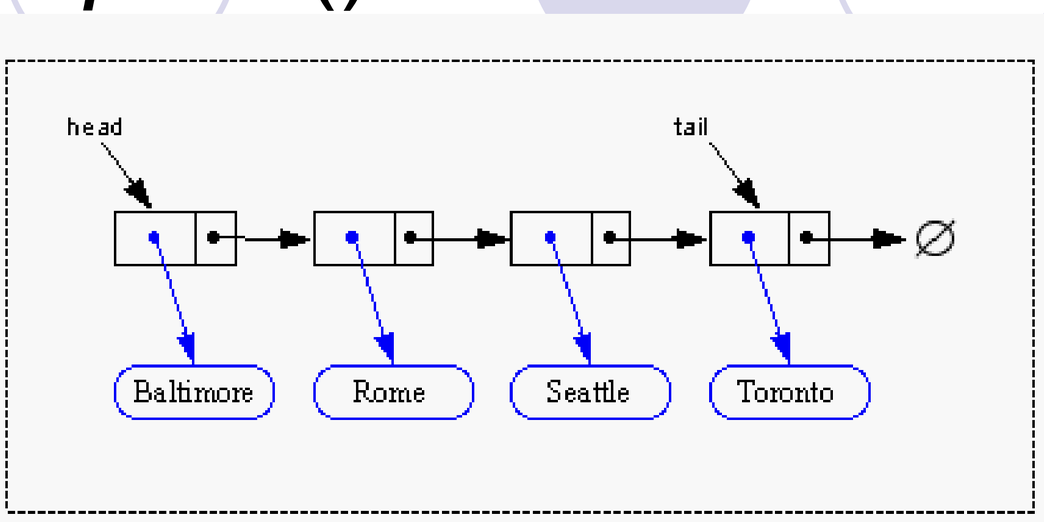
- Alternatives?
  - Extendable arrays
  - Linked lists (singly or doubly linked???)

Implementing a Stack with a Singly Linked List

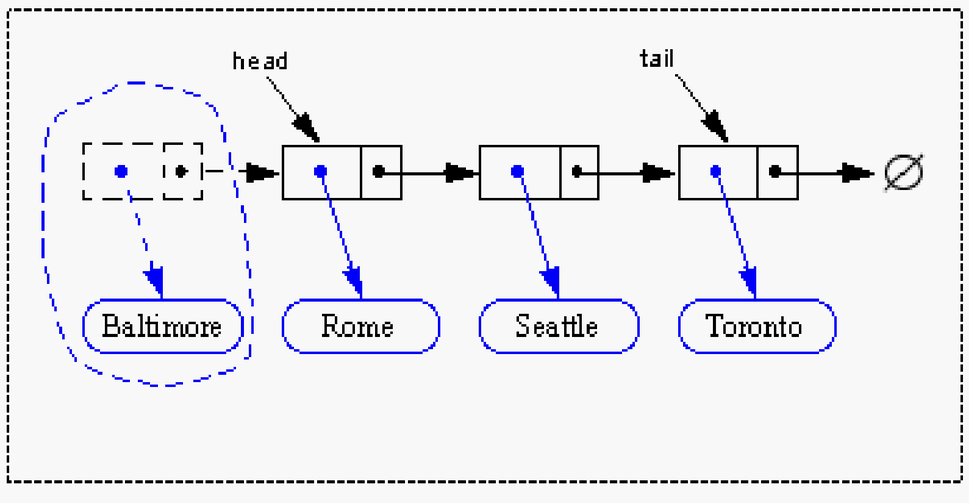
- Each node has two parts:
  1. pointer to the object (data stored)
  2. pointer to the next node in the list
- First node = head
- Last node = tail (*next* pointer is set to null)



pop() and *push*() Methods



- advance head reference



- inserting at the head is just as easy

Analysis of Linked List Implementation

- Space usage: $O(n)$
  n = number of elements in the stack
- Each operation runs in $O(1)$ time
- No limit on the stack size, subject to available memory
  (run-time error *OutOfMemoryError*)

Homework and Questions

- Implement the Stack ADT using singly linked lists
- List-based and array-based operations all run in $O(1)$ time. List-based implementation imposes no limit on the stack size, while array-based implementation does. Is list-based implementation better?
- Can we perform *push*() and *pop*() at the tail of the linked list? Analyze the running time in this case.

Next time ...

- Queues (5.2)